下载APP

22 | SQLite文本数据库：如何进行数据管理（下）？

2021-04-01 尹会生

Python自动化办公实战课 进入课程 >

讲述：尹会生
时长 22:23 大小 20.51M

你好，我是尹会生。

在上节课，我提到了使用比较简单的 SQL 来操作 SQLite，并为你讲解了数据库的基本操作步骤。

不过当你的程序功能越来越强大的时候，随之而来的就是代码的复杂度越来越高。像是上一讲，我们在进行 SQLite 数据库搜索的时候，你需要建立连接、申请游标对象，才能进行查询。而这些准备工作，我们更希望在程序运行的时候就准备好，这样就不必多次重复编写。

而且对数据库进行增删改查能够通过尽可能少的 SQL 来实现数据库的操作。那么能实现这一功能的就是**类**。

通过类，你可以为越来越复杂的程序编写结构更清晰的代码。同时也能更好地把 SQLite 的增删改查封装成一个独立的对象，便于你调用数据库时能进行数据持久化。

那么今天这节课，我就带你使用类来实现 SQLite 数据的读取和写入。与此同时，我会继续以通讯录为例，来给你讲解，如果使用了比较复杂的 SQL 来操作 SQLite 时，怎么合理组织代码结构，让你更优雅地书写代码。

使用类实现 SQLite 的读写

由于类这个概念比较抽象，我还是采用老办法帮你理解它，我将使用“类”对 SQLite 的读写 SQL 操作进行封装，并将类进行实例化以后进行调用，得到 SQLite 中的通讯录数据。我先把代码贴出来，供你参考：

复制代码

```
import sqlite3
import pathlib

class OptSqlite(object):
    def __init__(self, dbname = "new.db"):
        """
        :param dbname  数据库名称
        """
        self.dir = pathlib.PurePath(__file__).parent
        self.db = pathlib.PurePath(self.dir, dbname)
        self.conn = sqlite3.connect(self.db)
        self.cur = self.conn.cursor()

    def close(self):
        """
        关闭连接
        """
        self.cur.close()
        self.conn.close()

    def get_one_phone(self, username):
        """
        获取一个联系人的电话
        """

        self.get_user_phone_sql = f"""
            SELECT phone FROM address_book WHERE name = "{username}" """
        try:
            self.result = self.cur.execute(self.get_user_phone_sql)
            return self.result.fetchone()
        except Exception as e:
```

```
            print(f"失败原因是: {e}")

    def set_one_phone(self, name, phone):
        """
        增加一个联系人
        """
        self.set_user_phone_sql = '''INSERT INTO address_book
          VALUES (?, ?, ?)'''
        self.v =  (2, str(name), int(phone))
        try:
            self.cur.execute(self.set_user_phone_sql, self.v)
            self.conn.commit()
        except Exception as e:
            print(f"失败原因是: {e}")

if __name__ == "__main__":

    my_query = OptSqlite("contents.db")

    my_query.set_one_phone("Jerry","12344445555")

    phone = my_query.get_one_phone("Tom")
    phone2 = my_query.get_one_phone("Jerry")

    my_query.close()

    print(phone)
    print(phone2)

# 输出结果
# (12377778888,)
# (12344445555,)
```

在这段代码中，我使用类实现了两个连续操作：添加新的联系人“Jerry”，并取出联系人“Tom”和“Jerry”的手机号码。

通过代码，你会发现类的实现思路和语法，跟函数有非常大的区别，因此在你第一次使用类代替函数实现通讯录时，我要通过实现方式和语法方面来为你做个详细的对比，并且为你讲解类的初始化函数，在类实例化时是如何实现接收参数并自动初始化的。

总体来说，与使用函数实现数据库操作相比，类的最大优势就是完善的封装。

在使用类实现“SELECT”和“INSERT”这两个 SQL 操作的时候，你只需进行了一次初始化和关闭连接，后续的 SQL 操作都可以复用这次的连接，类能有效减少重复建立连接和重复初始化的工作。

因此在类似数据库封装这种功能复杂的代码中，你会看到更多的人选择用类代替自定义函数，实现开发需求。

从具体来讲，对比函数，类除了在封装方式上不同、语法和调用方式都不相同，我还是基于通讯录代码的封装和调用，为你讲解一下它和自定义函数的三个主要区别。

类和自定义函数的区别

首先，类和函数第一点区别就在于它们的对代码的封装方式上不同。

编写自定义函数，它的实现思路是通过函数去描述程序运行的过程，比如：代码的下一步需要做什么、需要什么参数。

而编写基于类的程序，它的实现思路更多要关注**相同的一类数据**，都有哪些属性和相同的动作。比如在代码中，我把数据库作为了一个类，因为类具有数据库名称这一属性，也具有查询和写入数据两个动作。而类在语法层面上，对属性和动作的封装要比函数更加完善。

在我工作中对建立数据库连接，以及执行查询、关闭数据库连接上都做过运行时间的测试，最终得出的结论是频繁地建立、关闭会给数据库带来较大的资源开销。因此，我在工作中会经常使用类把建立连接和关闭分别封装在多个查询动作之前和之后，确保这两个动作在多次查询时只执行一次，减少资源开销。

其次它们的语法结构也不同。函数是通过“def”关键字定义的，而类是通过“class”关键字定义的。

在编写一个新的类时，Python 语法还强制要求它必须继承父类，例如，我在编写的数据库类“OptSqlite”，就继承了父类“object”。继承父类意味这你可以在当前类中执行父类定义过的方法，而不需要再重新去编写一个定义过的方法。那如果你不需要继承其他类呢？这时候你就可以使用 object 作为你自定义类的父类使用。

同时，object 的关键字可以和定义类语法的“()”一起省略掉，因此你会看到其他人的代码出现，会有下面两种不同的写法，但含义却 (在 Python3.x 版本) 是完全相同的。我将两种写法写在下面供你参考。

复制代码

```
class OptSqlite(object):
class OptSqlite:
```

最后它们的调用方式也不同。这一点主要表现在各自成员能否被访问和运行方式两方面。

类的定义中，可以定义当前类的属性和方法。属性就是类具有的数据状态，方法就是类对数据可以执行哪些操作。

在类中，可以设置哪些属性和方法能够被类以外的代码访问到，比如：我定一个了“鸟”类。并且定义了它的属性是黄色，它的动作是可以飞、可以叫。那么你可以借用变量这种形式来实现鸟类的属性，借用函数的形式实现鸟类能飞、能叫的动作。

此外，在定义属性和方法时，你还能限制它们的访问范围。像函数的调用，你只能访问它的函数名称和参数、中间的变量是不能被函数外的程序访问的。

是否能访问，在计算机中也被称作作用范围。在这一方面，类要比函数拥有更灵活的作用范围控制。

那在执行方式，类也和函数不同。函数执行时可以直接使用函数名 + 括号的方式调用它，如果需要多次执行可以使用变量存放多次执行的结果。

而类在执行时，一般要进行实例化。例如鸟类，在需要使用时，会实例化为一个对象“鸟001”，对象就具有类的所有属性和方法。当你需要多次使用鸟类时，可以多次将鸟类实例化成不同的小鸟。

再回到通讯录的代码。类似的在通讯录的代码中，我将 SQLite 数据库定义为类以后，如果你的工作需要一个通讯录，就实例化一次。实例化之后的代码我单独拎了出来，如下：

复制代码

```
my_query = OptSqlite("contents.db")
```

如果需要多个通讯录，就把它实例化多次，并指定不同的 SQLite 数据库即可。每个数据库实例，都会有一个“get_one_phone()”方法和一个“set_one_phone()”方法，来实现

通讯录中联系人的读取和写入。

而为了表示属性和方法是在实例化中使用的，你还需要对它增加 self 关键字，即：使用实例的属性时，要用“self. 属性”的写法。使用方法时，要是将实例的方法第一个参数设置为 self，代码为“方法 (self)”。

类能够在封装和调用上提供比函数更灵活的方式，因此你会发现当功能复杂，代码数量增多了以后，很多软件都采用了类方式实现代码的设计。

类中的特殊方法“init”

在类中，有一个内置的方法叫做“**init**”，它叫做类的初始化方法，能实现类在执行的时候接收参数，还能为类预先执行变量赋值、初始化等，实现在类一运行就需要完成的工作。

“**init**()”方法的作用有两个，分别是：

1. 为实例接收参数；
2. 实例化时立即运行该方法中的代码。

当一个类实例化时，它可以像函数调用一样，接收参数。类实例化时，它的后面需要增加括号“()”，括号中可以指定实例化的参数。这个参数将交给“**init**()”方法，作为“**init**()”方法的参数，进行使用。

我来为你举个例子，来说明类是如何实现接收参数的。例如我在通讯录的例子中，实例化一个 SQLite 的“OptSqlite”类，实例化的代码如下：

复制代码

```
my_query = OptSqlite("contents.db")
```

这段代码中的“OptSqlite”就是类的名称，而“contents.db”是该类初始化时，输入的参数，也是 SQLite 数据库文件的名称。

要想实现“my_query”实例在“OptSqlite”类实例化时获得参数，就需要在类中使用初始化方法：

复制代码

```
def __init__(self, dbname = "new.db"):
```

在这段代码中，我定义了“**init**()”方法，并指定它的参数“dbname”之后，那么实例“my_query”就能够得到参数 dbname 变量的值“contents.db”了。

这就是一个实例化一个类，并如何在第一时间获得参数的完整过程。不过获得参数之后，你还要对参数继续使用和处理，以及需要在实例化之后就立即运行一些代码，这些功能就可以写在“**init**()”方法中来实现。

例如我就将数据库文件的路径处理、初始化连接、初始化游标的代码写入到了初始化函数。代码如下：

复制代码

```
class OptSqlite(object):
    def __init__(self, dbname = "new.db"):
        """
        :param dbname  数据库名称
        """
        self.dir = pathlib.PurePath(__file__).parent
        self.db = pathlib.PurePath(self.dir, dbname)
        self.conn = sqlite3.connect(self.db)
        self.cur = self.conn.cursor()
```

通过上面的写法，实例不但能够接受参数，还能在初始化时做很多主要逻辑前的预备操作。这些初始化操作让实例被调用时的主要逻辑更加清晰。

为了能够让你对类有更深刻的理解，也为了能让你将数据库的代码直接拿来在工作中使用，我们在对数据库的写入和读取基础上，再增加修改和删除功能，这样，SQLite 的类就能完整实现数据库的增删改查功能了。

使用类实现完整的 SQLite 增删改查

SQLite 的增删改查，都需要依赖 SQL 语句完成，在编写代码前，我们先来学习一些更新和删除的 SQL，在掌握增删改查 SQL 基础上，你会更好地理解我编写操作 SQLite 类的代码逻辑。

更新和删除记录的 SQL 语句

首先，我先来带你学习一些更新的 SQL 语句。更新一般是对单个记录进行操作，因此更新的 SQL 语句会带有筛选条件的关键字“WHERE”。以更新“Tom”手机号码的 SQL 语句为例，我将更新需要用到的 SQL 语句，单独写出来供你参考：

```
1 UPDATE address_book SET phone=12300001111 WHERE id=1;
```

复制代码

在这条 SQL 语句中：

“UPDATE”是指即将更新的数据表。

“WHERE”是指更新的条件，由于“id”的主键约束条件限制，它的值在这张表中是唯一的，因此通过“WHERE id=1”会读取该表的“id”字段，得到唯一的一条记录。

“SET”用于指定记录中的“phone”字段将被更新的具体值。

这就是更新语句的各关键字的作用，那我们再来看看删除操作的 SQL 语句。例如我希望删除通讯录中的“Jerry”用户，就可以使用如下的 SQL 语句。

```
1 DELETE FROM address_book WHERE id=1;
```

复制代码

在这条 SQL 语句中，“DELETE FROM”用于指定表，“WHERE”用于指定过滤条件。

我想你肯定还发现了，无论更新还是删除操作中，都包含了“WHERE”关键字。使用了“WHERE”关键字，也就意味这“UPDATE 和 DELETE”也读取了数据库。因此，我们将插入和删除也称作是“使用 SQL 语句对数据库执行了一次查询”。当你为以后工作中编写复杂的“UPDATE 和 DELETE”语句时，如果遇到它们的性能达不到你预期的要求，可以从“查询”方面先对你的 SQL 语句进行优化。

在你对 SQL 语句不熟练的时候，我有一个建议提供给你，由于 UPDATE 和 DELETE 语句在没有指定条件时，会将整张表都进行更新和删除，所以我建议你在编写代码时，先通过

SELECT 得到要操作的数据，再将 SELECT 改写为 UPDATE 或 DELETE 语句，避免因手动操作失误导致数据发生丢失。

接下来我们就把修改和删除功能也加入到“OptSqlite”类中，实现对数据库的增删改查操作。

实现增删改查的类

实现了增删改查的“OptSqlite”类代码如下：

复制代码

```
import sqlite3
import pathlib

class OptSqlite(object):
    def __init__(self, dbname = "new.db"):
        """
        :param dbname  数据库名称
        """
        self.dir = pathlib.PurePath(__file__).parent
        self.db = pathlib.PurePath(self.dir, dbname)
        self.conn = sqlite3.connect(self.db)
        self.cur = self.conn.cursor()

    def close(self):
        """
        关闭连接
        """
        self.cur.close()
        self.conn.close()

    def new_table(self, table_name):
        """
        新建联系人表
        """

        sql = f'''CREATE TABLE {table_name}(
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            phone INT NOT NULL
            )'''

        try:
            self.cur.execute(sql)
            print("创建表成功")
        except Exception as e:
            print("创建表失败")
```

```
            print(f"失败原因是: {e}")

    def get_one_phone(self, username):
        """
        获取一个联系人的电话
        """

        self.get_user_phone_sql = f"""
            SELECT phone FROM address_book WHERE name = "{username}" """
        try:
            self.result = self.cur.execute(self.get_user_phone_sql)
            return self.result.fetchone()
        except Exception as e:
            print(f"失败原因是: {e}")

    def get_all_contents(self):
        """
        取得所有的联系人
        """
        try:
            self.result = self.cur.execute("SELECT * FROM address_book")
            return self.result.fetchall()
        except Exception as e:
            print(f"失败原因是: {e}")

    def set_one_phone(self, name, phone):
        """
        增加或修改一个联系人的电话
        """
        if self.get_one_phone(name):
            self.set_user_phone_sql = '''UPDATE address_book
            SET phone= ? WHERE name=?'''
            self.v =  (int(phone), str(name))
        else:
            self.set_user_phone_sql = '''INSERT INTO address_book
            VALUES (?, ?, ?)'''
            self.v =  (None, str(name), int(phone))
        try:
            self.cur.execute(self.set_user_phone_sql, self.v)
            self.conn.commit()
        except Exception as e:
            print(f"失败原因是: {e}")

    def delete_one_content(self, name):
        """
        删除一个联系人的电话
        """
        self.delete_user_sql = f'''DELETE FROM address_book
                WHERE name="{name}"'''

        try:
            self.cur.execute(self.delete_user_sql)
```

```
            self.conn.commit()
        except Exception as e:
            print(f"删除失败原因是：{e}")

if __name__ == "__main__":

    # 实例化
    my_query = OptSqlite("contents.db")

    # 创建一张表
    # my_query.new_table("address_book")

    # 增加或修改一个联系人的电话
    my_query.set_one_phone("Jerry","12344445556")

    # 查询一个联系人的电话
    phone = my_query.get_one_phone("Jerry")
    print(phone)

    # 查询所有人的电话
    contents = my_query.get_all_contents()
    print(contents)

    # 删除一个联系人
    my_query.delete_one_content("Jerry")

    contents = my_query.get_all_contents()
    print(contents)

    # 关闭连接
    my_query.close()
```

在这段代码中，实现的主要逻辑，是将代码的相似功能尽量封装成一个方法，将数据库初始化连接放在“**init**()”方法，并尽量复用这个连接。为此，我编写类“OptSqlite”实现通讯录操作的时候，使用了四个方法，我按照这四个方法在代码里的定义顺序依次为你分析一下。

第一个方法是创建通讯录的数据表。我把创建通讯录数据表的功能定义成类的一个方法。定义类的方法我刚才已经教过你了，它是借用函数的语法格式来定义的。

不过我在定义通讯录表的时候，还对 id 这个主键增加了一个新的修饰条件，叫做**自增“AUTOINCREMENT”**，它的用途是每插入一条记录，它的值就会自动+1。“SQL92 标准”中规定自增只能修饰整数类型的主键，所以我把 id 的类型改为“INTEGER”，否则在创建表时，SQLite 会提示类型不符合要求而报错。

第二个方法是查看通讯录所有的联系人。这和我们学习过的查看单个联系人时，使用的“SELECT 某个字段”在 SQL 语句是有区别的。当你需要匹配所有字段时，不用把所有字段逐一写在“SELECT”SQL 语句后面，你可以使用“*”来代替所有的字段，这样实现起来更便捷。

此外，在查询结果上面，由于 fetchone() 函数只返回多个结果中的第一条，因此我把它改为 fetchall() 函数，这样就能把查询到的所有联系人都显示出来。

而且 Python 比较友好的一点是，它会把整个通讯录显示为一个列表，每个联系人显示为元组，联系人的各种属性都放在相同的元组中，方便你能对取出来的数据再次处理。它的执行结果是：

复制代码

```
[(1, 'Tom', 12344445555), (2, 'Jerry', 12344445556)]
```

第三个方法是更新用户手机号码，由于更新操作的 UPDATE 语句和新增操作 INSERT 语句，对通讯录这一场景，实现起来非常相似。因此我没为它们两个功能编写两个方法，而是都放在了同一个方法 -- “set_one_phone()”方法中了。

这样做的好处是，使用“set_one_phone()”方法的人不用区分联系人是否存在，如果用户不存在，则通过条件判断语句，使用“INSERT”语句新建一个联系人。如果联系人存在，则改用“UPDATE”语句更新联系人的手机号码。

第四个方法是删除某个联系人，使用的是“DELETE”SQL 语句。由于这里的 SQL 语句拼接比较简单，我没有单独使用一个变量 v 来保存，而是使用了 f-string 字符串把变量直接替换到字符串中，拼接为一个 SQL 语句。

对于以后工作中遇到的简单的字符串替换，你也可以采用这种方式，会对代码阅读上带来比较流畅的阅读体验。

通过这四个方法，我实现了“OptSqlite”类的增删改查功能。实例化“OptSqlite”类之后，你只需了解每个方法的名称和参数，就能利用我编写的四个方法实现通讯录的完整操作。这也是采用类替代了函数实现更完善的封装，最大的优势。

小结

最后，我来为你总结一下本讲的主要内容。我们在这节课第一次编写了基于类的代码。通过对比类和函数的差别，我们了解到类的编写方法。这些差别体现在如何定义类、类中的成员属性和方法、以及一个用于接收参数、在实例化类时完成初始化的特殊方法“**init**()”。当你接触更多的其他人编写的 Python 代码时，就会慢慢发现代码量较大的程序，都会采用基于类的方式封装代码。也希望你在掌握类之后能够通过读懂其他人的代码，对自己的编码能力进行提升。

此外，我还用类重新封装了基于 SQLit 的通讯录的基本功能，其中就包括增删改查。相信你在掌握了对数据库的封装之后，可以把原有需要用 SQL 与数据库打交道的接口，封装为类的方法，这样也有助于你能够把 SQLite 更多的应用于自己的办公优化中来。

思考题

按照惯例，最后我来为你留一道思考题，在本讲的代码中，我使用“INSERT”增加联系人之前没有判断该联系人是否存在。你能否利用判断语句实现增加联系人前对联系人是否存在进行判断，并提示用户对重复联系人进行合并操作呢？

欢迎把你的思考和想法放在留言区，我们一起交流讨论。如果这节课学习的数据透视表对你的工作有帮助，也欢迎你把课程推荐给你的朋友或同事，一起做职场中的效率人。

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | SQLite文本数据库：如何进行数据管理（上）？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。